UNIVERSIDAD AUTONOMA DE MADRID

ESCUELA POLITECNICA SUPERIOR

Grado en Ingeniería de Tecnologías y Servicios de Telecomunicación

TRABAJO FIN DE GRADO

Desarrollo mediante lenguaje de alto nivel de un sistema de simulación de redes basado en FPGA para aplicaciones multiGbps Ethernet

Alfredo de Lema Sorrosal
Tutor: Gustavo Sutter

Febrero 2016

Desarrollo mediante lenguaje de alto nivel de un sistema de simulación de redes basado en FPGA para aplicaciones multiGbps Ethernet

AUTOR: Alfredo de Lema Sorrosal
TUTOR: Gustavo Sutter

High Performance Computing and Networking Research Group
Dpto. de Tecnología Electrónica y de las Comunicaciones
Escuela Politécnica Superior
Universidad Autónoma de Madrid
Febrero de 2016

Resumen

El objetivo de este proyecto es crear una aplicación capaz de simular el comportamiento de una red de 10 Gbps con el objetivo de testar equipamiento que se conecta a una red a estas velocidades.

Disponer de herramientas de simulación o emulación del tráfico real de red es de gran utilidad para poder comprobar o verificar el uso de nuevos protocolos y equipos en un entorno que pueda parecerse a las condiciones que soporta una red real.

Actualmente existen herramientas Software ejecutándose sobre hardware de propósito general que pueden emular el tráfico como es el caso de la conocida herramienta Netem [Lin15]. Esta herramienta es ampliamente utilizada en entornos de baja velocidad (hasta un Gbps) pero tienen serias limitaciones cuando las tasas de línea se incrementan (multi-Gbps) y se quiere tener precisiones internas que estén por debajo del microsegundos.

La solución para soportar altas tasas de línea con bajas latencias prescindiendo de software corriendo sobre un sistema operativo es realizar una solución hardware. Por cuestiones de coste de desarrollo y flexibilidad la solución es el uso de circuitos reconfigurables tipo FPGAs (*Field Programmable Gates Arrays*) [Qui15] que brindan precisiones del orden de nanosegundos y tasas de multiGpbs.

La solución HW basado en tarjetas de FPGAs garantiza tanto alta velocidad como baja latencia pero la programación tradicional basada en lenguajes de descripción de hardware (HDL – *Hardware Description Languajes*) como VHDL o Verilog es sumamente costosa en términos de esfuerzo y tiempo de desarrollo. Para mitigar el esfuerzo de desarrollo han aparecido en el mercado aproximaciones basadas en lenguaje de alto nivel lo que deriva en la noción de síntesis de circuitos desde alto nivel (HLS – *High Level Synthesis*).

Concretamente este proyecto creará un modelo funcional con características reducidas de la aplicación Netem utilizando lenguajes de alto nivel, concretamente se sintetiza con la herramienta Vivado-HLS [Xil15a], para ser utilizado en tarjetas basadas en dispositivos FPGAs. El sistema es capaz de recibir paquetes a 10Gbps, asociarles un retardo pseudoaleatorio y una tasa de pérdida de paquetes generando el tráfico de salida con el objeto de emular una red de paquetes.

Palabras clave

FPGA, NetFPGA, Netem, Red a 10 Gbps, Lenguaje de alto nivel (HLS), Lenguaje de descripción hardware(HDL), Vivado, Vivado HLS, Ethernet, Emulador de tráfico de red, Cores, BlockRAMs, Testbench, Pipeline.

Abstract

The main goal of this project is to create an application capable to simulate the behavior of a 10 Gpbs network in order to test equipment which is connected to a network at these speeds.

The possibility to have simulation or emulation tools for real network traffic is useful to check the use of new protocols or equipments in an environment that may resemble conditions that supports a real network.

Currently there are software tools running on general-purpose hardware that can emulate the traffic such as the well-known Netem [Lin15] tool. This tool is widely used in low-speed (up to Gbps) but has serious limitations when line rates are increased (multi-Gbps) and when internal delay are below the microseconds.

The solution to support high line rates with low latency is using dedicated hardware instead of software running on an operating system. For cost and flexibility reasons the solution is the use of reconfigurable circuits like FPGA (Field Programmable Gates Array) [Qui15] providing nanoseconds precition and multiGpbs rates.

The hardware solution based on FPGAs boards guarantees both speed and low latency but traditional programming based on hardware description languages (HDL) such as VHDL or Verilog is costly in terms of effort and development time. To mitigate the development cost, have appeared on the market approaches based on high level language which leads to the notion High Leven Synthesis (HLS).

Specifically this project will create a functional model of Netem application using High Level Languages (HLL), specifically synthesized with Vivado-HLS [Xil15a] tool targeting to use in FPGAs boards. The system is able of receiving packets at 10 Gbps, associating a pseudorandom delay and loss rate packets generated outbound traffic in order to emulate a packet network.

Keywords

FPGA, NetFPGA, Netem, 10 Gbps Network, High Level Synthesis (HLS), Hardware Description Languaje(HDL), Vivado, Vivado HLS, Ethernet, Network Traffic Emulator, Cores, BlockRAMs, Testbench, Pipeline.

Agradecimientos

Me gustaría dedicar este trabajo fin de grado a todas las personas que me han apoyado y siempre han estado cerca a lo largo de la carrera. Una carrera que no ha sido nada fácil y realmente imposible sin vosotros.

En primer lugar, me gustaría dar las gracias a mi tutor Gustavo Sutter por toda la ayuda y dedicación realizada a lo largo de este proyecto, una ayuda que me ha permitido aprender muchas cosas durante este último año.

También dar las gracias a Isa, José, Rafa y Mario, compañeros del HPCN Lab que siempre me han ofrecido ayuda en momentos de dificultad.

Durante estos años de carrera me llevo una gran cantidad de amigos, gente maravillosa con los que he vivido grandes y malos momentos, gracias por haber estado siempre ahí porque sin vosotros esta no hubiera sido la mejor etapa de mi vida. En especial: Sergio, Eduardo, Tito, Ricardo, Ángel, Andrea, Alberto, Dani, Bárbara, Marta, Nacho, Raúl, Marta, Víctor, Darío y Xan.

Sobre todo dar las gracias a mi familia, es especial a mis padres: Alfredo y Gloria, y a mis hermanos: Daniel y Laura, por todos los valores y la educación que me habéis dado desde pequeñito, algo fundamental para convertirme en la persona que soy ahora. Siempre habéis estado en todo momento para apoyarme y salir adelante en los momentos más difíciles.

Quiero agradecer a Sara, mi novia, todo el apoyo que me ha dado este añito. Un año que no ha sido nada fácil pero ha sido más llevadero junto a ti.

Detrás de cada obra hay un trasfondo, y en ese trasfondo hay unos compañeros de vida que siempre han estado ahí. Dar las gracias a Alex, Pepe y Yeyo, mis grandes amigos de la infancia.

Gracias de corazón.

INDICE DE CONTENIDOS

INDICE DE FIGURAS

INDICE DE TABLAS

1 Introducción

1.1 Motivación.

Desde un principio Internet ha sido considerado como una infraestructura de investigación, actualmente es un ambiente completamente operativo donde resulta difícil llegar a realizar pruebas.

Desde universidades, empresas o departamentos de investigación que trabajan en redes necesitan de herramientas para poder comprobar o verificar el uso de nuevos protocolos que se asemejen a las condiciones reales que soporta una red. En la mayoría de los casos, debido al gran coste computacional, estas comprobaciones son difíciles de realizar.

Existen varias soluciones de abordar este problema. Por un lado tenemos las herramientas de simulación como pueden ser la herramienta de **Netem** [Lin15], de la que más adelante hablaremos, la cual permite diversas medidas controladas a bajo coste. Entre sus principales inconvenientes es lo que comentábamos anteriormente, la gran carga computacional que debe de soportar, especialmente para un gran número de fenómenos. Por otro lado, normalmente los emuladores de red suelen abreviar el sistema a medir, dejándolo algo fuera de la realidad.

Por último, y principal inconveniente de la emulación de sistemas de red mediante uso de software es la limitación de la tasa de línea. Netem está limitado por lo que si queremos simular una **red a 10 Gbps** [Wik15] tendremos un grave problema. Y menos aún para futuras redes entre 40-100 Gbps.

Este es el momento donde entra en juego la utilización de hardware para la simulación de una red real, que será de lo que consistirá este proyecto. Entre los principales problemas de la utilización de hardware será el elevado coste de las máquinas.

El uso de **FPGAs** [Qui15] es una gran elección de coste moderado para la resolución de estos problemas hardware que con tecnologías de circuitos específicos (ASICs) serían algo impensable por problemas de coste.

Para la realización de este proyecto se hará uso de lenguajes de alto nivel, por medio de la plataforma **Vivado HLS [**Xil15a**]** desarrollada por Xilinx donde por medio de lenguaje C++ podremos generar cores IP para ser enviados a cualquier programa de Xilinx, en nuestro caso **Vivado [**Xil15b**].** De esta manera, se facilitará a la hora de programar o debuguear en comparación a hacerlo directamente mediante lenguaje de descripción hardware (**HDL – Hardware Description Language**).

1.2 Objetivos

El principal objetivo de este proyecto es la realización de un prototipo que sea capaz de emular el tráfico de red de una red a 10 Gbps bajo el punto de vista de nivel Hardware, mediante lenguaje de alto nivel (HLS).

1.3 Organización de la memoria

La memoria se estructura de la siguiente manera, se describirá desde el estado del arte hasta el proceso de desarrollo del proyecto, pasando por la arquitectura propuesta hasta las pruebas realizadas. Finalmente se verán las conclusiones y posibles trabajos futuros.

En el primer capítulo se ha realizado una breve introducción del proyecto, la motivación y los objetivos del mismo.

En el segundo capítulo se expondrán en el estado del arte las principales bases de este proyecto, pasando desde lo que son las redes a 10 Gbps, los principales emuladores de red comerciales, así como las FPGAs y diseño usando HLS.

En el tercer capítulo se verá la arquitectura propuesta, pasando desde las pruebas realizadas, análisis de las latencias y mejoras de las mismas y terminando con un estudio de las posibles necesidades de requerimientos en casos reales.

En el cuarto capítulo se evaluarán los resultados obtenidos en el capítulo anterior, viendo las limitaciones del prototipo y haciendo una evaluación de otras posibilidades.

Finalmente, en el quinto capítulo se explicarán las conclusiones y posibles trabajos futuros.

2 Estado del arte

2.1 Redes a 10 Gbps

Las actuales redes a 10 Gbps están basadas en el estándar 10 Gb Ethernet, conocido técnicamente como 802.1 ae (2002) [Wik15], desarrollado por el grupo IEEE y usa velocidades diez veces más rápidas que su antecesor, Gigabit Ethernet 1Gbps.

Este cambio de estándares es debido a la creciente demanda de ancho de banda en la red, causado por aplicaciones y servicios que requieren un uso intensivo del ancho de banda ya que cada vez hay mayor número de dispositivos conectados a la red como pueden ser los teléfonos inteligentes "smartphone", televisiones inteligentes o tablets.

Tales aplicaciones como pueden ser video IP o la evolución de servicios basados en la nube para clientes de empresas o residenciales son las principales causas del aumento del tráfico de red hoy en día.

El estándar 10 Gigabit Ethernet no solo aumenta la velocidad a 10 Gbps manteniendo sus propiedades, sino que es capaz de incrementar la distancia de interconexión hasta los 40 km, por lo que es aplicable a redes de área amplia (WAN).

Primeramente 10 Gb fue ideado para funcionar sobre fibra óptica, pero IEEE logró que este nivel fuese compatible bajo los cables de cobre, aumentando así su compatibilidad y aceptación comercial, por lo que no hay que preocuparse por las infraestructuras ya instaladas del antiguo protocolo minimizando gastos.

Para el modelo OSI, Ethernet pertenecería a los niveles 1 y 2. Dicho protocolo 10 GbE conserva las propiedades principales de la arquitectura Ethernet, desde el formato de las tramas, tamaños e el protocolo de acceso al medio.

Visto esto, podríamos decir que la tecnología 10 GbE es la más extendida para redes LAN de alta velocidad, presentando las siguientes ventajas:

- Comparado con las distintas opciones del mercado, presenta menores costes tanto de mantenimiento como de instalación.

- Las infraestructuras actuales de Ethernet son fácilmente implementadas bajo 10 GbE.

- Admiten la reutilización de protocoles, herramientas y procesos ya realizados en la infraestructura de administración.

- Y por último y más importante, de aumentar la velocidad nominal de transmisión en diez veces más respecto a su antecesor, lo que provoca un gran salto en muchos de los procesos que hoy reclaman las primordiales aplicaciones del mercado.

A medida que este tipo de redes va aumentando es necesario tener herramientas de emulación de tráfico de red para simular las distintas situaciones desfavorables que pueden ocurrir en una red de área amplia luego este tipo de redes a 10 Gbps serán con las que se trabajaran durante este proyecto.

2.2 Emuladores de tráfico de red

La función de un emulador es tratar de componer de forma precisa un dispositivo o sistema de tal forma que éste actúe como si estuviera siendo usado en el aparato original, de esta forma simular solamente trata de imitar el comportamiento del programa.

Por lo tanto, hará falta hacer uso de un emulador de gran confianza que se acerque en gran medida a la realidad, en el que pueda modificarse según a las necesidades de los distintos escenarios que se desee emula.

A continuación se exponen varios emuladores de tráfico de red comerciales a nivel software, donde principalmente nos centraremos más en Netem.

2.2.1 Netem

Netem [Lin15] proporciona la funcionalidad de emulador de red para realizar distintas pruebas de protocolos, emulando las características de una red de área amplia (WAN – Wide Area Network).

El gran motivo del uso de Netem es suministrar una manera de poder reproducir grandes redes en un ambiente de laboratorio. El primer uso conocido de Netem fue el de probar nuevas mejoras de implementación de TCP en Linux.

Netem se maneja mediante la línea de comando, concretamente con el comando tc, que forma parte del conjunto de herramientas del paquete iproute2.

Un ejemplo de la configuración de Netem sería el siguiente:

tc qdisc ***add*** *dev <interfaz> root netem <parámetro> <valor>*

Donde podemos encontrar varios tipos de variables.

Por un lado el parámetro puede adoptar las siguientes adaptaciones según el valor requerido.

- *Delay*: Para especificar un retardo requerido en <valor>.

- *Loss*: Para establecer una serie de porcentaje de pérdidas en <valor>.

- *Corrupt*: Para poder elegir en <valor> un determinado porcentaje de paquetes corruptos.

- *Duplicate*: Donde especificaremos un porcentaje de paquetes duplicados en <valor>.

- *Reorder*: Para especificar en <valor> un porcentaje de paquetes para desordenar.

Por otro lado existen otra serie de comandos de gran uso como son los siguientes:

- Para poder cambiar un parámetro ya configurado podemos hacer uso ***change***
  *tc qdisc **change** dev <interfaz> root netem <parámetro> <valor>*

- Para monstrar los parámetros ya definidos podemos hacer uso de ***show***
  *tc qdisc **show***

-Y por último para borrar los parámetros de una interfaz con ***del***
  *tc qdisc **del** dev <interfaz> root*

Entonces Netem es una herramienta con gran utilidad para reproducir las circunstancias que afectan a las transmisiones de redes bajo el protocolo TCP haciendo uso de herramientas estadísticas indispensables para hacer una emulación de cómo responde una red en escenarios reales.

2.2.2 Otros emuladores de tráfico de red

A parte de Netem, existen muchos más emuladores de tráfico de red, entre los principales se encuentran los siguientes [Kri15].

-**NIST Net**: es una extensión del kernel de Linux que proporciona demoras, pérdidas y otras opciones de emulación. Al trabajar a nivel IP, puede emular condiciones características de una red WAN. Actúa sobre los paquetes entrantes antes de que aparezcan a la pila del protocolo y utiliza hardware de alta resolución de tiempos.

-**Umlsim**: es un híbrido, mitad emulador, mitad simulador, que usa Linux en modo usuario para hacer uso de una simulación dirigida por eventos, admite probar la pila del protocolo TCP/IP estándar utilizando un pseudo dispositivo que es capaz de simular una red.

2.3 FPGA aplicadas al procesamiento de redes

Un FPGA (*Field Programmable Gate Array*) es un circuito integrado que puede configurarse para poder realizar cualquier función lógica. Es un componente estándar reprogramable luego las entradas como las salidas deben de ser también reprogramables [Qui15].

Las FPGA nace en 1985 con una idea simple: un Gate Array tolerante a errores de diseño y reprogramable por el usuario. Es el resultado de la concurrencia de dos tecnologías, los dispositivos lógicos programables (PLDs) y los circuitos integrados de aplicación específica (ASIC).

El primer fabricante de las FPGAs fue Xilinx cuyos dispositivos siguen siendo uno de los más populares en las compañías y grupos de investigación. Otros fabricantes pueden Altera (adquirida por Intel), microsemi (antigua Actel) o Atmel.

Entre las principales características que podemos destacar de las FPGAs son:

- Fácil de depurar.
- Gran tolerancia a los fallos.
- Alta fiabilidad.
- Diseño reducido.
- Alta complejidad.
- Costes bajos en el desarrollo.

Las funciones lógicas se deben de mapear en los transistores como ocurre en los Gate Array, pero en el caso de las FPGAs como hemos mencionado antes, deben de ser reprogramables. Las funciones creadas por el usuario se mapearan mediante una tabla “look-up” pudiendo simular cualquier tipo de puerta lógica.

El uso de estos dispositivos al campo del procesamiento de redes es altamente difundido. Entre otros existen proyectos académicos de placas basadas en FPGA para el procesamiento de redes como NetFPGA [Kal12].

NetFPGA es un proyecto open HW & SW para el desarrollo de aplicaciones de red desarrollado por la U. de Stanford and U de Cambridge con el apoyo de Xilinx.

Luego NetFPGA es una plataforma para desarrollar software y hardware de código abierto para la creación rápida de prototipos de dispositivos de red de ordenadores.

Este proyecto va principalmente dirigido hacia usuarios de la industria, investigadores académicos o los propios estudiantes. NetFPGA utiliza un enfoque basado en FPGA para dispositivos de prototipos de red.

Existen múltiples versiones de tarjetas NetFPGA incluso tarjetas para procesar a 10 Gbps, como por ejemplo, tenemos el caso de la tarjeta NetFPGA SUME.

NetFPGA SUME [Dig15] es una tarjeta PCI Express basada en FPGA con capacidades I/O para operaciones de 10 a 100 Gbps, una tarjeta adaptadora PCIe Gen3 x8 incorporando una Virtex-7 690-T de Xilinx. SUME puede ser utilizado como NIC, como un conmutador multipuerto, firewall, como entorno de pruebas o mediciones, y muchos más entre otros.

Las principales ventajas de ello es permitir a los usuarios desarrollar diseños que son capaces de procesar paquetes a velocidad de línea, una capacidad que no es posible por enfoques basados en software.

Mediante una NetFPGA 10G son inmensos los usos que podremos sacar de ella gracias al potente procesador del que dispone, como por ejemplo permitiendo analizar una gran cantidad de número de paquetes. Esto será de gran utilidad para la emulación del tráfico de una red de 10 Gbps mediante uso de hardware.

2.4 Diseño usando síntesis de alto nivel (HLS)

Un lenguaje de descripción hardware (HDL) es un lenguaje de programación que se utiliza para definir la estructura, diseño y operación de los circuitos electrónicos o circuitos electrónicos digitales por lo que harían posible una explicación especifica de un circuito electrónico, haciendo posible un análisis y simulación del mismo. Los lenguajes HDL que dominan el mercado son VHDL [Syn97] y Verilog [Ovi96] y son una definición a nivel RTL (Register Transfer Level).

Estos lenguajes tienen dificultad a la hora de programar o debuguear, causa principal por la cual se hayan desarrollado distintas herramientas que permiten simplificar la programación de hardware mediante el uso de lenguajes de más alto nivel (HLS). Existen distintas herramientas para lenguaje de alto nivel (HLS – High Level Synthesis). En este proyecto se hará uso de la herramienta de Xilinx llamada **Vivado HLS [**Xil15a**]**.

Mediante Vivado HLS podremos acelerar la creación de módulos IP permitiendo ser descritos en lenguajes C y C++ para ser utilizados directamente a cualquier dispositivo de Xilinx. En particular el core generado se utilizará, en nuestro caso en la herramienta de diseño **Vivado** [Xil15b**]** integrando el diseño con otros módulos RTL. Vivado HLS proporciona sistemas y arquitecturas de diseño mediante el camino más rápido con la creación de los módulos IP.

Dentro de la herramienta de Vivado HLS, podremos comprobar el funcionamiento de cada uno de nuestros cores mediante simulaciones de *testbench* también descriptas C/C++ pudiendo de estar manera ir probando por partes el funcionamiento de todo el proyecto.

Por lo tanto, para el diseño usando HLS se hará uso de la herramienta Vivado HLS, en comparación a otras herramientas del mercado, por su rápido aprendizaje, el nivel de optimización y las comodidades de poder ver el comportamiento de cada módulo originado, como principales peculiaridades.

3 Arquitectura propuesta.

La principal finalidad de este proyecto es el diseño, desarrollo e implementación de un simulador de una red real a nivel hardware, por lo tanto, a continuación se expone una propuesta de la arquitectura que se llevará a cabo.

De breve introducción podemos decir que todos los paquetes que vayan llegando, entrarán por conexión AXI-STREAM, dichos paquetes se guardarán en bloques de memoria, en nuestro caso se utilizarán BlockRAMs [Xil05c], ya que la FPGA de Xilinx que se usará cuenta con ellas internamente y acelerará el proceso. Todo ello llevará asociado una marca temporal (time stamping) para saber cuándo entraron los paquetes y para saber cuándo será el momento de que deban salir. Adicionalmente el modulo tendrán en cuenta un retardo aleatorio asociado a cada paquete y un porcentaje de perdida. A continuación se describe en profundidad estos aspectos.

3.1 Uso de Memorias BRAMs

Con el objeto de almacenar los paquetes internamente y otros descriptores necesarios si utilizaran memorias internas de las FPGAs de Xilinx denominadas BRAM (o Block RAMs).
Estas memorias son de doble puerto y tamaño de 18Kbits pudiéndose configurar el ancho de los datos.

El motivo del uso de estos componentes es la alta velocidad de acceso (responden en un ciclo de reloj) y pueden operar a la frecuencia de la lógica programable.

Los dispositivos modernos de la serie-7 de Xilinx dependiendo del tamaño del dispositivo cuentan entre 100 y 5640 block RAMS. En particular el dispositivo utilizado (Xilinx Virtex-7 XC7V690T FFG1761-3) dispone de 2940 BRAMs, es decir en el orden de los 25 MB.

3.2 Descripción de los módulos usando HLS

La siguiente arquitectura, como hemos mencionado anteriormente, se realizará mediante la herramienta de síntesis de alto nivel **Vivado-HLS [**Xil15a**]** en la que se generarán unos cores que podrán ser descritos mediante lenguaje de alto nivel, es decir, en lenguaje C / C++, quitándose así gran complejidad de tener que ser descritos mediante lenguaje de descripción hardware. En concreto, se generarán tres cores los cuales serán descritos en el primer prototipo. Estos cores se conectarán mediante la herramienta de Xilinx **VIVADO [**Xil15b**]**, donde esta vez sí mediante lenguaje Hardware, en particular, de Verilog, se mapearán e interconectarán mediante el uso de BlockRAM.

Las BlockRAM serán del siguiente estilo, las cuales serán de doble puerto, es decir, desde ambos puertos se podrá leer y escribir. Una BlockRam funcionará como una memoria RAM en la que se irán introduciendo los datos mediante las direcciones por el puerto Addr, y mediante la activación de lectura o de escritura de los puertos En o Wen, se irán metiendo los datos por Din o leyendo los datos por Dout. Todo esto irá de manera síncrona por la función de los puertos Clk.

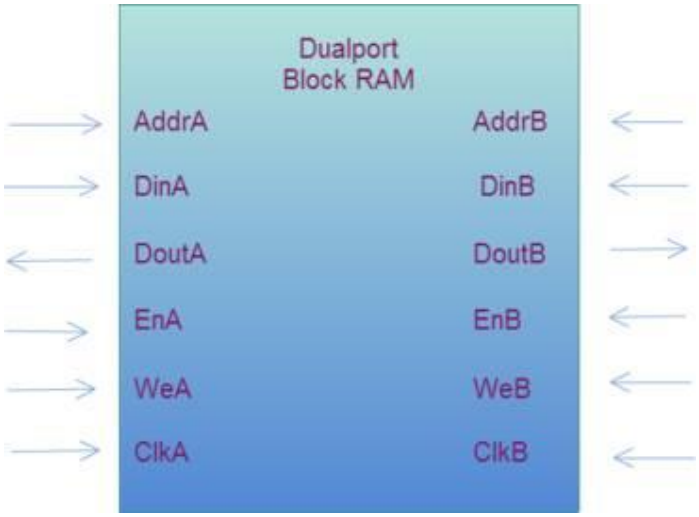

Figura 3.1: Arquitectura de una BlockRAM

El uso de BlockRAM es una gran elección para la interconexión y sincronización de los cores.

3.3 Primer prototipo.

Para este primer prototipo, se expone la siguiente arquitectura:

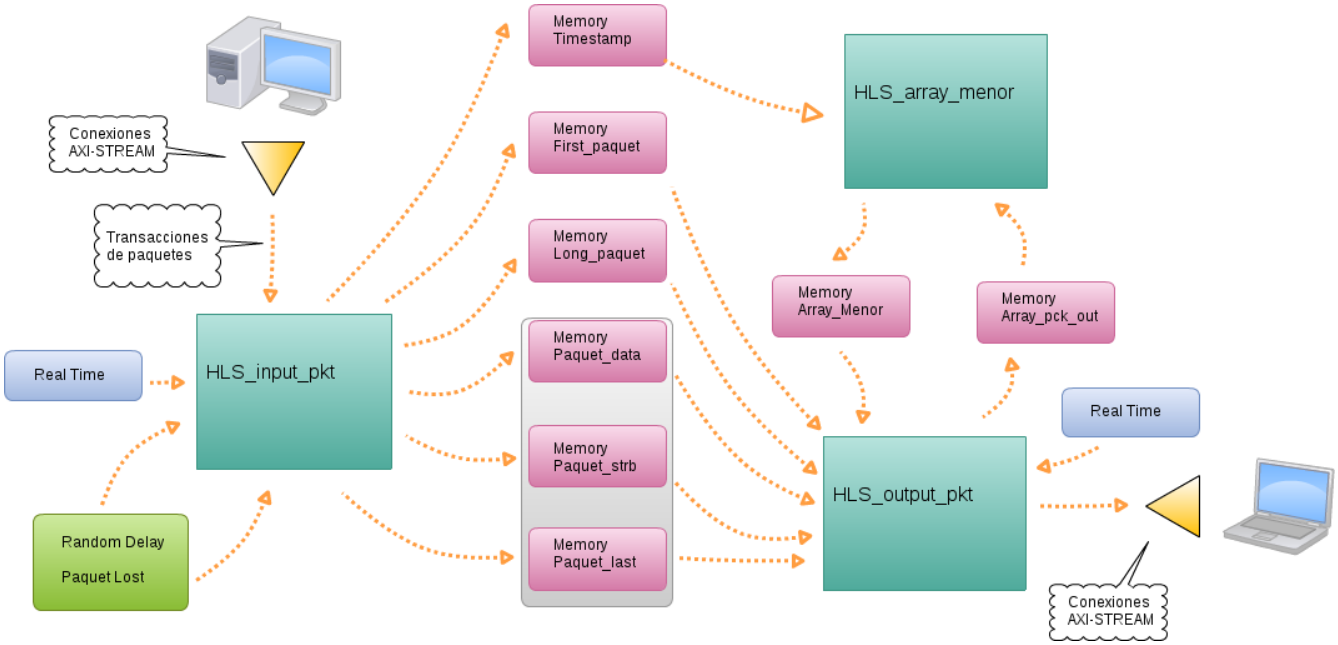

Figura 3.2: Arquitectura general del primer prototipo.

Cada módulo de color verde, es un core generado por la herramienta VIVADO HLS, que como se comentó en el punto anterior, cada uno está conectado con diferentes memorias, para interconectar y hacer síncrono todo el proceso.

- En primer lugar, tenemos el módulo HLS_input_pkt en el que se recibirán los paquetes mediante una traza pcap, y mediante un stream tendremos los paquetes por el puerto input de dicho módulo. Dentro de este módulo se irán diseccionando los paquetes, sacando diferentes características de cada uno como cuál es el primer paquete de cada ráfaga de paquetes, cual es la longitud de cada uno o el respectivo timestamp, todo ello en función de los respectivos strb y last de los paquetes que recibamos.

En el caso de los timestamp, se les sumará a cada paquete un retardo aleatorio que se generará externamente, con el objeto de simular el comportamiento de una red real.

Estas características de los paquetes se guardarán en memorias auxiliares (los módulos de color rosa de la figura).

También está la opción, al igual que hablábamos de los retardos, de poner una serie de pérdidas de paquetes aleatorias según un porcentaje determinado, dichos paquetes no llegarán a guardarse en las memorias.
Por lo tanto, bajo el punto de vista del módulo HLS_input_pkt, solamente se escribirá en las memorias, nunca se leerá aunque estas sean de doble puerto de lectura/escritura.

- En segundo lugar, está el módulo HLS_array_menor. Dicho módulo, como se puede ver en la figura 3.2, estará conectado con la memoria de los timestamp, estos timestamp se irán leyendo a medida que vayan llegando, y como dijimos en el apartado anterior respecto a los retardos aleatorios de cada paquete, aquí se encargará de ir reordenando los timestamp para saber que paquete tiene el turno de salir.
  Por lo tanto, en la memoria Array_menor se escribirán los respectivos índices respecto a la ordenación de los paquetes según cada retardo.
  Por último, en la memoria Array_pck_out, se leerán los índices de los paquetes que ya han salidos para que en nuestro módulo no vuelvan a ser ordenados y así reducir el coste computacional.

- Por último, tenemos el módulo HLS_output_pkt donde se encargará de sacar los paquetes de las memorias anteriormente guardados en memoria Paquet_data, memoria Paquet_strb y memoria Paquet_last, según dos condiciones. Por un lado, que cumpla el tiempo de expiración del paquete según un contador global del fichero genérico de la herramienta VIVADO [Xil15b], y por otro lado, que sea su turno según los índices anteriormente guardados y que serán leídos en la memoria Array_menor.

Con este índice se sabra que paquete tendrá que salir mediante las memorias de First_paquet y Long_paquet.
Una vez haya salido dicho paquete, se reiniciarán los registros usados con un -1 para que puedan volver a ser usador para guardar nuevos paquetes.
A parte de reiniciar dichos registros, hay que avisar a la memoria Array_pck_out de que dicho paquete acaba de salir.

Como ya se comentó anteriormente, tanto los tres módulos HLS como el conjunto de las memorias, son mapeados en un fichero mediante la herramienta VIVADO [Xil15b] mediante el lenguaje de hardware Verilog, los cuales van acompañados con alguna pequeña lógica como el contador te tiempo de expiración, el contador global para los timestamp de los nuevos paquetes u otros pequeños detalles.

3.4 Pruebas del primer prototipo

Durante las pruebas de este primer prototipo, se llevará a cabo una simulación funcional del sistema.

Para ello mediante la herramienta de Xilinx, VIVADO, se comprobará el comportamiento de este primer prototipo mediante el uso de un testbech de un sistema completo en el que se mapeará todo el conjunto de los bloques mencionados anteriormente.

A partir de este punto, desde la herramienta Wireshark, se capturará una serie de tráfico que será exportado en formato pcap. Estas trazas pcap se llevarán a nuestro testbench para tener trenes de paquetes para realizar las pruebas de simulación.

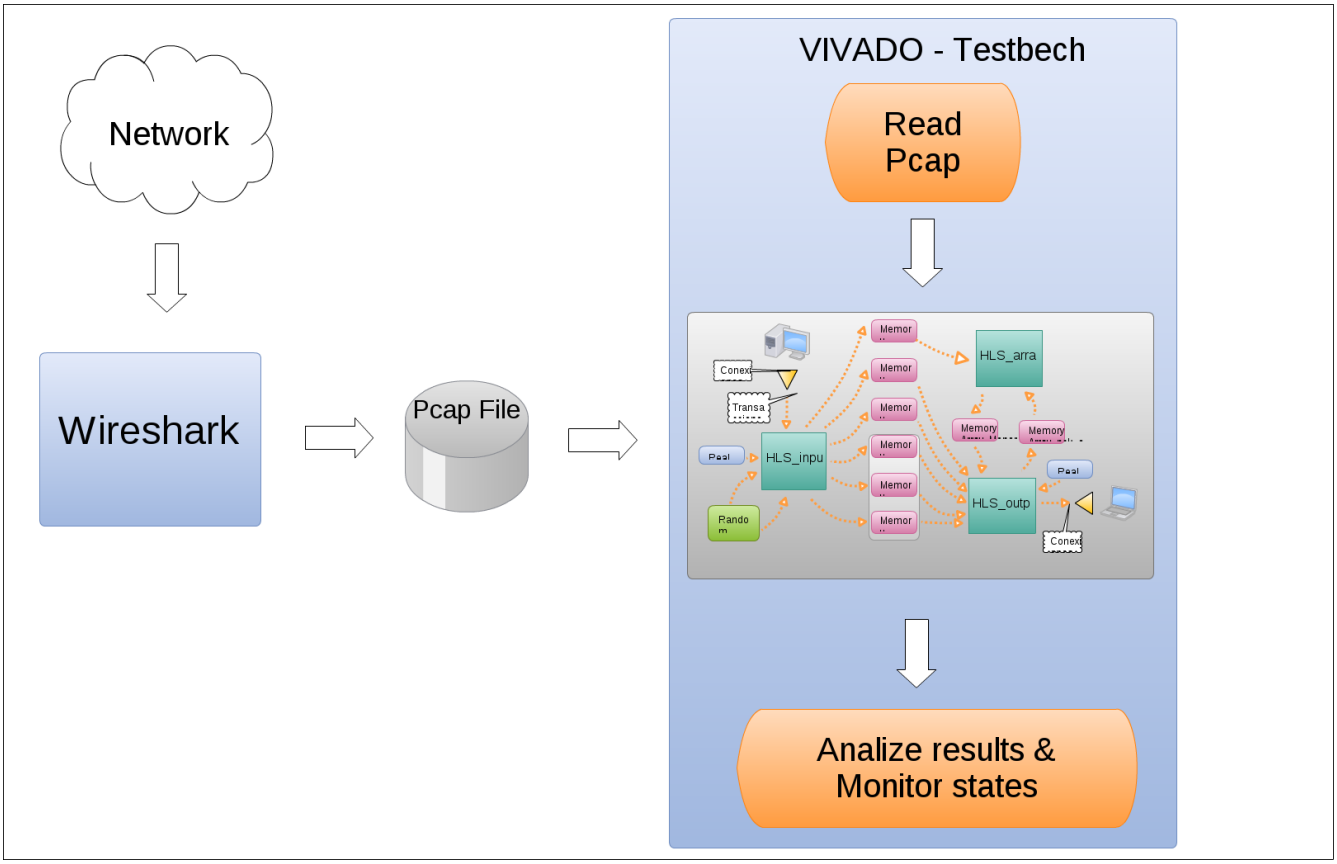

Figura 3.3: Esquema pruebas primer prototipo.

3.4.1 El banco de pruebas (testbench) del sistema

El testbench es capaz de leer un fichero en formato pcap y generar los estímulos para el circuito respetando la distancia que se lee del fichero pcap.

3.4.2 Pruebas realizadas

En primer lugar, se probará este primer prototipo con una traza pcap corta, de unos doce paquetes. En la siguiente simulación, dichos paquetes llegarán seguidos, sin espacios de tiempo entre ellos, a continuación se muestra una captura de la simulación.

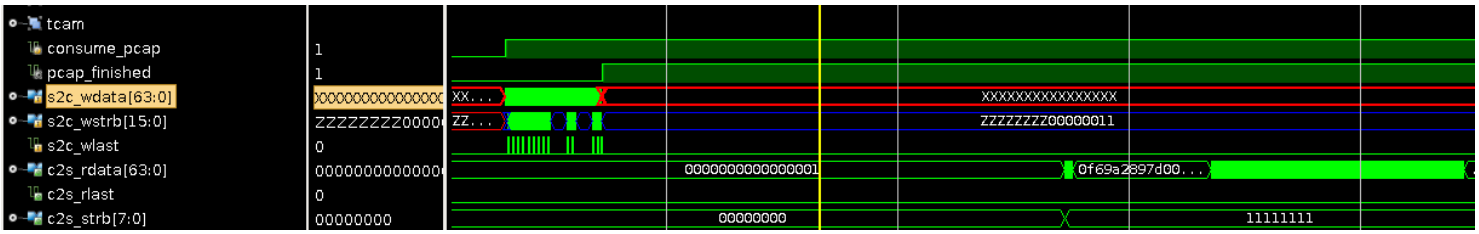

Figura 3.4: Simulación traza pcap corta sin separación entre paquetes.

Como se puede apreciar, hay un tiempo de retraso en el que comienzan a salir los paquetes. Esto es debido a que todos ellos llevan un retardo incorporado y tienen que ser ordenados por el módulo HLS_array_menor además de esperar su tiempo de vencimiento.

En segundo lugar, en la misma traza pcap anterior, se ha introducido una separación entre los paquetes según el timestamp de cada uno para simular un caso en que los paquetes llegasen por ráfagas y no todos al mismo tiempo.

Figura 3.5: Simulación traza pcap corta con separación entre paquetes.

Este sería un ejemplo que se asemejaría más a las redes reales que el anterior ya que las redes actuales están sometidas a grandes delays.

Por último, se muestra un caso parecido al anterior pero con una separación entre los paquetes del doble que el anterior.

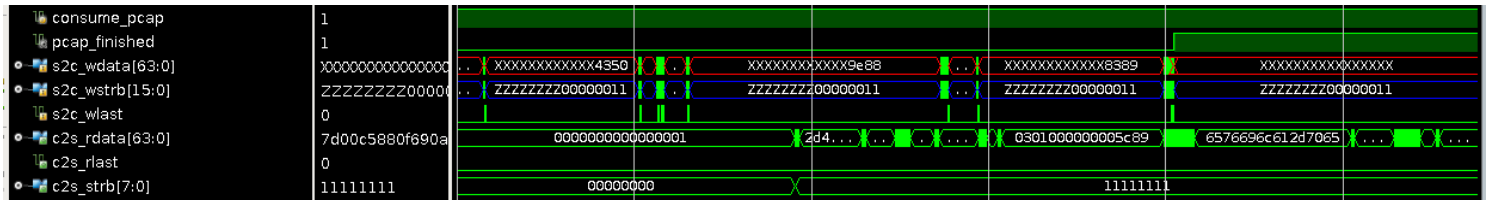

Figura 3.6: Simulación traza pcap corta con mayor separación entre paquetes.

Apreciando las tres simulaciones, podemos afirmar que cuanto mayor sea la separación entre los paquetes, mejor funcionará el prototipo ya que habrá más tiempo para que las memorias y los módulos HLS trabajen y no se saturen con la llegada de muchos paquetes al mismo tiempo.

Por otro lado, la siguiente simulación representa un tren de paquetes continuos, de unos cincuenta paquetes, dicha simulación se asemeja bastante a la primera anteriormente mostrada

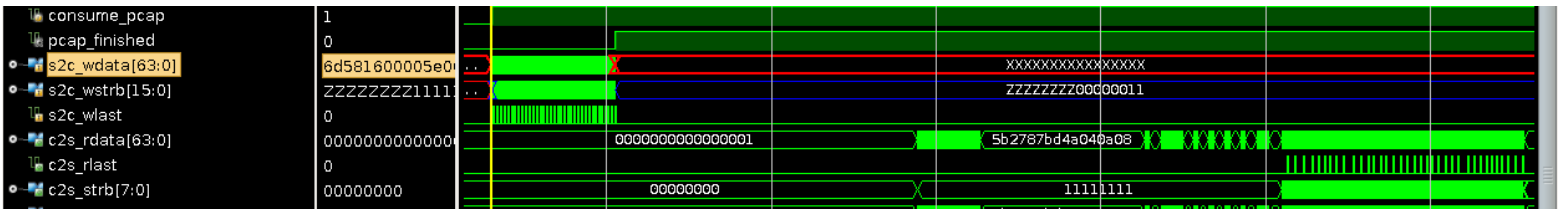

Figura 3.7: Simulación traza pcap larga sin separación entre paquetes.

Al igual que antes, la siguiente simulación representa el tren de paquetes de la simulación de arriba pero con una pequeña separación entre los mismos.

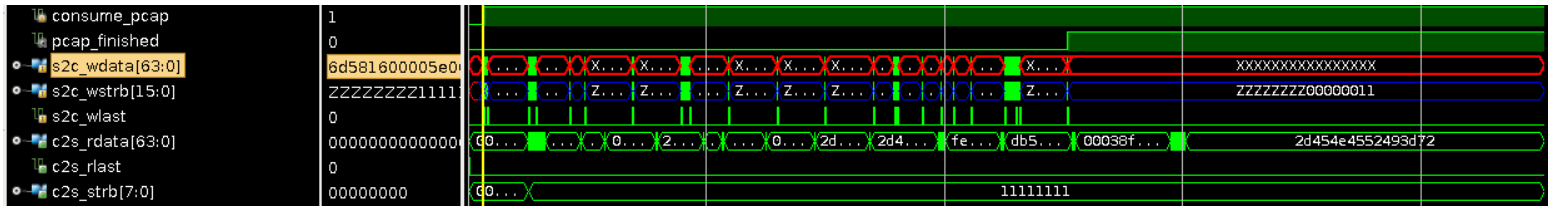

Figura 3.8: Simulación traza pcap larga con separación entre paquetes.

Para terminar, se muestra la siguiente simulación con el mismo número de paquetes que la anterior simulación pero con una gran separación entre los mismos.

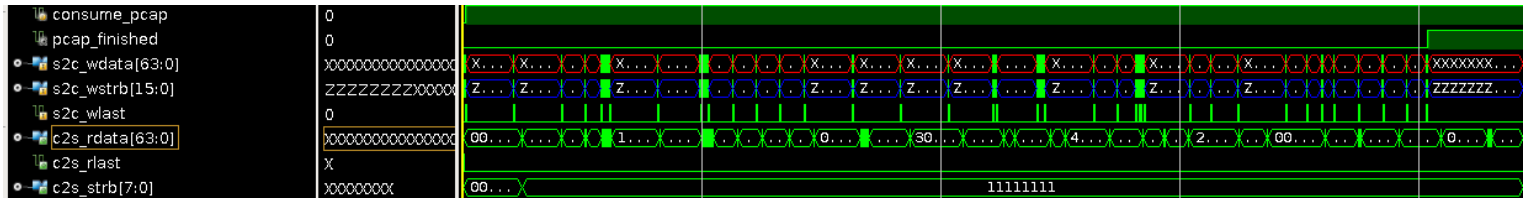

Figura 3.9: Simulación traza pcap larga con mayor separación entre paquetes.

Como se puede observar, las pruebas de simulación de este primero prototipo generan bastante latencia por lo tanto el siguiente paso será mejorar estas latencias para que se puedan soportar un flujo de paquetes de 10 Gbps.

En la siguiente y última captura de simulación, se muestra una de las principales limitaciones de este primer diseño.

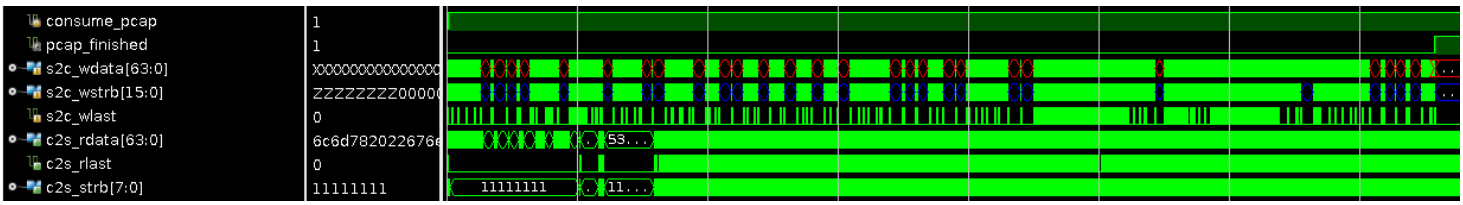

Figura 3.10: Simulación traza pcap de dos mil paquetes con pequeña separación.

Como podemos observar, nos llegan una gran cantidad de paquetes por nuestra interfaz de entrada y con poca separación entre los mismos. Bajo el punto de vista de las memorias, éstas son limitadas, se guardarán tal cantidad de paquetes que no dará tiempo a reorganizar los tiempos de retardo aleatorios de cada paquete y como consecuencia se sobre escribirán los datos de los paquetes. Por lo tanto, producto de las latencias internas se llenarán rápidamente las memorias de este primer prototipo.

Una solución a este problema puede ser o por un lado ampliar el tamaño de las memorias y por otro lado, exigir una separación entre los propios paquetes para que dé tiempo a nuestros módulos HLS a recalcular los tiempos y poder ordenar los paquetes según el retardo de los mismos.

En el caso del tamaño de las memorias que están en este primer prototipo, podemos observar en la siguiente captura como el siguiente Warning nos va avisando cuando las memorias están a punto de llenarse y por último, nos avisa de un error cuando dichas memorias se han llenado del todo.

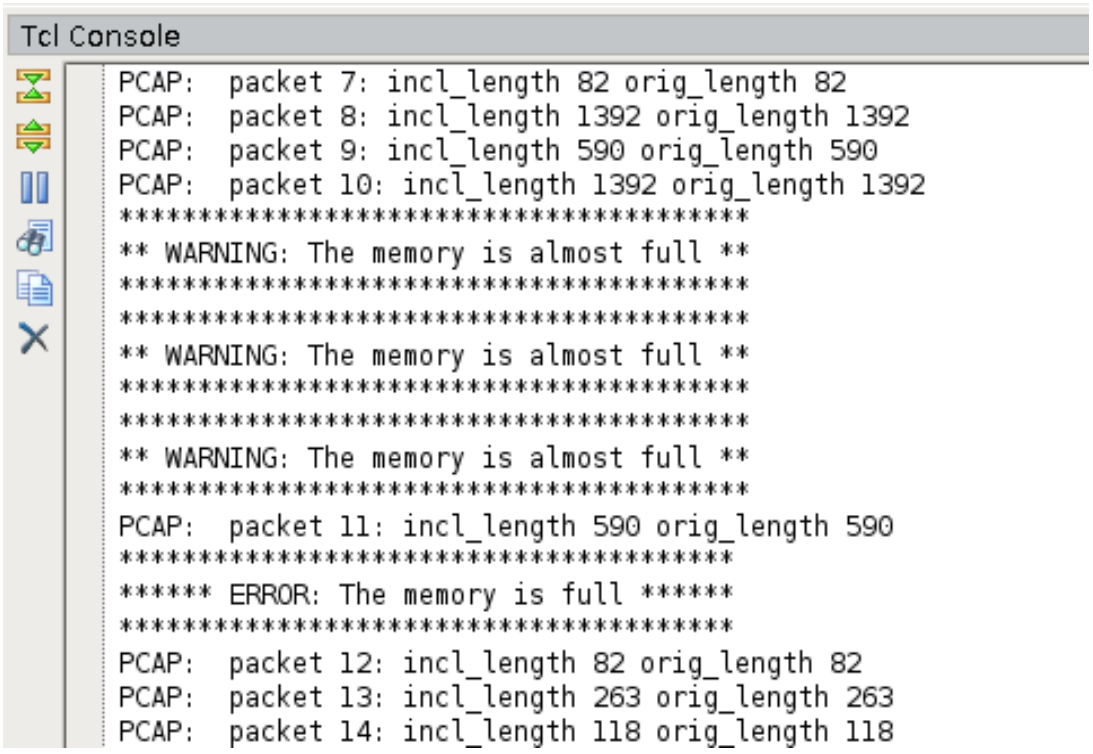

Figura 3.11: Captura de la consola para mostrar alertar (Warning) de llenado de memoria.

3.5 Análisis de latencias del proceso

Bajo el punto de vista de las latencias de este proceso podemos considerar tres fases para poder medirlas. En primer lugar, será calcular el tiempo de demora que tarda nuestro primer módulo HLS en leer el primer paquete.
Respecto a la segunda fase, tenemos nuestro módulo de índices, en el que estos tiempos dependerán del número de iteraciones que tenga que realizar según sean los valores aleatorios asociados a cada paquete. Y como última fase de latencias tenemos el módulo final, en el que se considerará el tiempo que tarda en salir cada paquete.

Para la primera fase, mediante la herramienta Vivado HLS, usando la línea de código *#pragma HLS LOOP_TRIPCOUNT min=10 max=100* se podrá saber en el número de ciclos que se realiza el código. Por lo tanto, en la siguiente figura se puede apreciar que la latencia en este caso es de dos, es decir, que cada vez que se ejecute este primer módulo, se necesitarán dos ciclos de reloj.

⊟ **Loop**

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- main_loop	20	200	2	-	-	10 ~ 100	no

Tabla 3.1: Latencia módulo HLS_input_pkt

A pesar de que todo el módulo se ejecuta mediante dos ciclos de reloj, en la figura 3.13, se puede observar como cada uno de los paquetes que se leen, son en un ciclo de reloj.

Por lo tanto como este módulo obtiene características de los paquetes, es la razón de que tarde dos ciclos, uno para leer los paquetes y otro para poner dichas características en las memorias.

Figura 3.12: Ciclos de reloj por cada paquete, módulo HLS_input_pkt

Respecto a la segunda fase, al igual que antes, con la herramienta Vivado HLS, y usando la misma línea de código, podremos saber el número de ciclos de reloj que hará falta para ejecutar el módulo al completo, en este caso el módulo es HLS_array_menor, dicho módulo será el que tenga que usar mayor número de ciclos de reloj ya que es el encargado de recalcular los índices y por tanto tendrá mayor número de operaciones.

Como se podrá observar en la siguiente figura, el módulo tiene varios bucles por lo tanto cada uno de ellos añade mayor latencia, y la latencia total del módulo será la suma de las latencias por lo tanto en esta caso la latencia total será de entre ocho y nueve.

⊟ Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- inic_loop	512	512	1	-	-	512	no
- main_loop	20	107040400	2 ~ 1070404	-	-	10 ~ 100	no
+ main_loop.1	20	200	2	-	-	10 ~ 100	no
+ aux_gen_l1	320	30200	32 ~ 302	-	-	10 ~ 100	no
++ aux_gen_l2	30	300	3	-	-	10 ~ 100	no
+ array_menor_l1_array_menor_l2	300	1040000	3 ~ 104	-	-	100 ~ 10000	no
++ array_menor_l1_array_menor_l2.1	10	100	2	-	-	10 ~ 100	no

Tabla 3.2: Latencia del módulo HLS_array_menor

Para la última fase, al igual de como se hizo en los puntos anteriores se muestra la siguiente figura en la que como se puede apreciar cada la latencia en este módulo será de cuatro, por lo tanto cuatro ciclos de reloj cada vez que se tenga que ejecutar este módulo.

Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- Loop 1	512	512	1	-	-	512	no
- main_loop	?	?	?	-	-	10 ~ 100	no
+ bucle_output	?	?	4	-	-	?	no

Tabla 3.3: Latencia del módulo HLS_output_pkt

Por otro lado, en la siguiente figura, siendo esta una captura de simulación se puede apreciar como por cada paquete que sale de este módulo se necesitan cuatro ciclos de reloj.

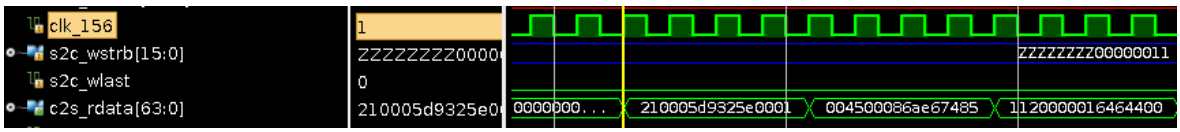

Figura 3.13: Ciclos de reloj por cada paquete, módulo HLS_output_pkt

Una vez analizado las latencias de los módulos HLS en su primer prototipo, en el punto siguiente se verá cómo se pueden reducir estos ciclos de reloj en cada uno de los módulos, es decir, poder optimizarlos para que hagan su misma función en unos ciclos menos de reloj.

3.6 Mejoras en latencia

En este apartado, se tratará de optimizar los módulos HLS de manera que se reduzcan el número de ciclos de reloj durante la ejecución. Por tanto, antes de todo hay que hablar de lo que significa hacer un *Pipeline* de un proceso.

La segmentación o *Pipeline* del inglés, es un método en electrónica en el que se es capaz de mejorar el rendimiento de ciertos sistemas electrónicos digitales, mayoritariamente se usa en microprocesadores. La arquitectura en *Pipeline* reside en ir cambiando un flujo de datos en un proceso comprendido por varias fases secuenciales, siendo la entrada de cada una la salida de la anterior.

Una vez se ha tenido una idea general de lo que consiste la arquitectura *Pipeline*, se irá módulo por módulo viendo la optimización que se pueda realizar en cada uno de ellos.

Al igual que en el punto anterior, mediante la herramienta de Xilinx, Vivado HLS se empezará por el módulo HLS_imput_pkt, donde mediante la siguiente línea de código *#pragma HLS PIPELINE* para aplicar la arquitectura *Pipeline* y con #pragma HLS LOOP_TRIPCOUNT min=10 max=100 para saber el número de ciclos de reloj que usará nuestro proceso se tendrán los siguientes resultados.

⊟ **Loop**

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- main_loop	11	101	2	1	1	10 ~ 100	yes

Tabla 3.4: Latencia del módulo HLS_input_pkt optimizado

Como se puede apreciar en la figura de arriba, el módulo se ha optimizado en un ciclo, por lo tanto se ha reducido en un ciclo ya que antes todo ello tardaba dos ciclos de reloj, al igual que se puede ver como en el apartado de *Pipeline* nos marca que ha sido aplicado.

En segundo lugar, respecto a la optimización del módulo HLS_array_menor, se ha conseguido reducir en gran medida el número de ciclos de reloj de este core, por un lado planteando el código de otra manera, con el resultado de optimizar la latencia y por otro lado aplicando la directiva *#pragma HLS PIPELINE*. En la figura a continuación, se muestra como hay menor cantidad de bucle que en la figura 3.14, por lo tanto la latencia será bastante menor.

Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- inic_loop	512	512	1	-	-	512	no
- main_loop	30	10500	3 ~ 105	-	-	10 ~ 100	no
+ array_menor_l2	10	100	2	1	1	10 ~ 100	yes

Tabla 3.5: Latencia del módulo HLS_array_menor optimizado

En el caso del módulo HLS_output_pkt, siguiendo el mismo método que en el primer core y aplicando la arquitectura *Pipeline* donde sea posible aplicarla, se muestra la siguiente figura.

Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- Loop 1	512	512	1	-	-	512	no
- main_loop	20	1600	2 ~ 16	-	-	10 ~ 100	no
+ bucle_output	2	10	2	1	1	2 ~ 10	yes

Tabla 3.6: Latencia del módulo HLS_output_pkt optimizado

Comparando esta figura con la figura 3.15, se puede observar como mediante la arquitectura Pipeline hemos podido reducir los ciclos de reloj desde cuatro hasta uno. Respecto al main_loop, sigue igual que antes ya que no siempre se puede aplicar este tipo de optimización.

3.7 Estudio de las necesidades de velocidad y almacenamiento en casos reales

En la siguiente sección se analizarán en función de los resultados anteriores cuales serían los requerimientos realistas para una red de 10 Gbps. Se expondrá un escenario de baja, media y alta carga.

Para un escenario de media o alta carga, estarán llegando continuamente paquetes cortos, en donde habrá que averiguar la máxima tasa de llegada, tratar de saber en qué ventana de tiempo podríamos analizar cuantos paquetes llegarían a estar almacenados en las memorias del sistema e intentar saber a qué velocidad debería de funcionar todo en ciclos de reloj.

Dichos escenarios dependerán en gran medida del número de paquetes que lleguen desordenados, ya que esto influenciará a la hora de recalcular los tiempos y por tanto tendrá mayor tiempo de demora, y si los paquetes son recibidos continuamente, debemos de tener unas memorias de gran tamaño.

Por otro lado, bajo un escenario en el que no hay apenas retardos entre los paquetes y nos llegan prácticamente en orden, necesitaremos unas memorias bastante menores ya que apenas habrá que recalcular los tiempos.

A continuación es expone la estructura de una trama Ethernet que servirá de ayuda para la tabla de más adelante.

Figura 3.14: Estructura trama Ethernet

A dicha figura, habría que sumarle otros valores opcionales como son entre Source Address y el Type, la etiqueta que es de 4 Bytes y por otro lado, después de FCS, también está el Gap entre frames que es de 12 Bytes. Por tanto, toda la trama Ethernet puede oscilar desde 84 Bytes hasta los 1542 Bytes.

Se expone a continuación la siguiente tabla en la que se reúne los valores de requerimientos necesarios y realistas.

Descripción	Resultados	
Máxima cantidad de paquetes a 10 Gbps. Paquetes cortos 42 bytes payload (peor escenario) (1)	14.880.952 paquetes/segundo 596,04 Mbytes/s= 5.1 Gbps	
Máxima cantidad de paquetes a 10 Gbps. Paquetes largos 1500 bytes payload (mejor escenario) (2)	810.635 paquetes 1,13 Gbytes/s= 9.06 Gbps	
Cantidad de paquetes a 10 Gbps en trazas reales. Paquetes de 900 bytes de payload y de 350 bytes de payload(valores realistas) [Cai14] (3)	1.326.963 paquetes 900 bytes de payload 1,11 Gbytes/s= 8,88 Gbps 3.378.378 paquetes 350 bytes de payload 1,04 Gbytes/s= 8,32 Gbps	
Tamaño de almacenamiento 1 uSec para paquetes cortos 46 bytes payload (peor escenario) (4)	14,88 paquetes 5,58 Kb de paquetes 0,465 Kb de índices	
Tamaño de almacenamiento 1 uSec para paquetes largos 1500 bytes payload (peor escenario) (5)	< 1 paquete (0,81) 9,525 Kb de paquetes 0,794 Kb de índices	
Tamaño de almacenamiento 1 uSec (valores realistas) (6)	Para 900 bytes payload: 1,359 paquetes 9,31 Kb de paquetes 0,776 Kb de índices	Para 350 bytes payload: 3,378 paquetes 8,72 Kb de paquetes 0,727 Kb de índices
Tamaño de almacenamiento 1 mSec para paquetes cortos 46 bytes payload (peor escenario) (7)	1.489 paquetes 5,45 Mb de paquetes 0,454 Mb de índices	
Tamaño de almacenamiento 1 mSec para paquetes largos 1500 bytes payload (peor escenario) (8)	810,6 paquetes 9,257 Mb de paquetes 0,771 Mb de índices	

Tamaño de almacenamiento 1 mSec (valores realistas) [Cai14] (9)	Para 900 bytes payload: 1.359,695 paquetes 9,09 Mb de paquetes 0,757 Mb de índices	Para 350 bytes payload: 3.378,378 paquetes 8,52 Mb de paquetes 0,71 Mb de índices
Valores realistas para enlaces 10Gbps (10)	2.368.536 paquetes/segundo	
Tiempo máximo de transmisión almacenable en XC690 para paquetes de 46 bytes de payload (11)	8,53 ms de tráfico es posible almacenar usando todas las BRAMs del dispositivo.	
Tiempo máximo de transmisión almacenable en XC690 para paquetes de 350 bytes de payload (12)	5,45 ms para llenado de memorias BRAMs	
Tiempo máximo de transmisión almacenable en XC690 para paquetes de 900 bytes de payload (13)	5,11 ms para llenado de memorias BRAMs	
Tiempo máximo de transmisión almacenable en XC690 para paquetes de 1500 bytes de payload (14)	5,02 ms para llenado de memorias BRAMs	

Tabla 3.7: Valores para los requerimientos necesarios en casos reales.

Cálculos realizados:

(1): CantPaquetes = velocTrans / TamTotalPaquetes

(10 bps * 10^9) / (84 bytes * 8 bits/byte)=14.880.952 paquetes/s

VelocidadEfectivaTransmisión = CantPaquetes * InformaciónEfectiva

14.880.952 paq/s*42 bytes/paq= 684523792 bytes/s /1024 kbytes/bytes = 668480 kbytes/s / 1024 Mbytes/kbytes= 596,04 Mbytes/s

(2): Mismo sistema que (1)

(10 bps * 10^9) / (1542 bytes* 8 bits/byte) = 810.635 paquetes

810.635 paq/s * 1500 bytes/paq= 1215952500 bytes/s /1024 = 1187453 /1024 = 1159/1024 = 1,13 Gbytes/s

(3): Mismo sistema que (1)

(10 bps * 10^9) / ((42+900 bytes)* 8 bits/byte)=1.326.963 paquetes/s

1.326.963 paq/s * 900 bytes/paq= 1194267516 bytes/s /1024 = 1166276 /1024 = 1138,94/1024 = 1,11 Gbytes/s

(10 bps* 10^9) / ((42+350 bytes)* 8 bits/byte)=3.188.775,51 paquetes

3.188.775,51 paq/s * 350 bytes/paq= 1116071428 bytes/s /1024 = 1089913,5 /1024 = 1064,37/1024 = 1,04 Gbytes/s

(4): Datos a almacenar= nroPaquetes * Tiempo * TamAlmac

14.880.952 paq/s * (10^-6 uSec/s)*48(i) bytes/paq * 8 bits/byte= 5714,28 bits

5714,28 /12(ii)= 476,19 bits

(i) Las transacciones son múltiplo de 8 y de esta manera se almacenan. Por tanto los 46 bytes se almacenan en 48 bytes.

(ii) De media, cada doce paquetes se guarda uno de los índices.

(5): Mismo sistema que (4)

810.635 paq/s * (10^-6 uSec/s)*1504(i) bytes/paq * 8 bits/byte= 9753,6 bits
9753,56 /12= 812,8 bits
(i) Las transacciones son múltiplo de 8 y de esta manera se almacenan. Por tanto los 1500 bytes se almacenan en 1504 bytes.

(6): Mismo sistema que (4)

1.358.695 paq/s * (10^-6 uSec/s) * 904(i) bytes/paq *8 bits/byte =9534,82 bits
9534,82/12= 794.57 bits
3.378.378 paq/s * (10^-6 uSec/s) * 352(i) bytes/paq *8 bits/byte =8933,53 bits
8933,53 /12= 744,46 bits

(7): Datos a almacenar = nroPaquetes * TamAlmac

14.880.952 paq/s * (10^-3 uSec/s)*48(i) bytes/paq * 8 bits/byte= 5.714.285,28 bits
5.714.285,28 /12(ii)= 476.190,44 bits
(i) Las transacciones son múltiplo de 8 y de esta manera se almacenan. Por tanto los 46 bytes se almacenan en 48 bytes.
(ii) De media, cada doce paquetes se guarda uno de los índices.

(8): Mismo sistema que (7)

810.635 paq/s * (10^-3 uSec/s)*1504(i) bytes/paq * 8 bits/byte= 9.706.626,1 bits
9.753.560,3 /12= 808.885,5 bits
(i) Las transacciones son múltiplo de 8 y de esta manera se almacenan. Por tanto los 1500 bytes se almacenan en 1504 bytes.

(9): Mismo sistema que (7)

1.358.695 paq/s * (10^-3 uSec/s) * 904(i) bytes/paq *8 bits/byte =9.534.827,4 bits
9.534.827,4 /12= 794.568,9 bits
3.378.378 paq/s * (10^-3 uSec/s) * 352(i) bytes/paq *8 bits/byte = 8.933.532 bits
8.933.532 /12= 744.461 bits

(10): ValoresRealistas= (MaxNumReal + MinNumReal)/2

(3.378.378+1.358.695)/2= 2.368.536 packets

(11): Considerando una Virtex 7 de modelo XC7VX690T, tendremos 51,68Mb de memorias BRAMs, por tanto si para 1 ms del (7), se ocupa 5,45 Mb, en 9,48 ms, se ocuparán los 51, 68 Mb de memoria de BRAMs, pero considerando que hay que reservar el 10% de las BRAMs para índices, FIFOs y demás, habrá que aplicar el 90% al tiempo anterior luego 8,53 ms.

(12): Mismo sistema que (9), para 51,68Mb de memoria BRAMs con paquetes de 350 bytes de payload se ocupan 8,52 Mb en 1ms, luego quedará lleno en 6,06 ms y aplicando el 90%, será de 5,45 ms.

(13/14): Mismo sistema que (9), para 51,68Mb de memoria BRAMs con paquetes de 900/1050 bytes de payload se ocupan 9,09/9,257Mb en 1ms, luego quedará lleno en 5,68/5,583 ms y aplicando el 90%, será de 5,11/5,02 ms.

4 Evaluación de resultados

Se hizo un demostrador capaz de hacer las funcionalidades de Netem en Hardware, pero tiene una serie de limitaciones que se verán en el punto siguiente.

4.1 Limitaciones

Haciendo referencia al estudio realizado en la sección 3.7 sobre las necesidades de velocidad y almacenamiento en casos reales, este prototipo dista de poder realizar una emulación realista de un entorno de red complejo funcionando a 10 Gbps. Los retardos en redes WAN están en el orden de los varios milisegundos pudiendo con este prototipo en casos extremos solo almacenar algunos pocos ms. No obstante en entornos LAN o de conexión dentro de equipos si alcanzará la cantidad máxima de datos almacenables.

Entre las mayores limitaciones caben destacar dos de ellas, en primer lugar se hablará sobre las memorias y en segundo lugar, se hará un pequeño estudio del core HLS_array_menor.

Como principal motivo, necesitaríamos memorias internas de un gran tamaño en el caso de que tengamos bastante volumen de tráfico. Por lo tanto, este prototipo solamente sería válido para casos en los que haya poco volumen de tráfico haciendo que no se saturen las memorias internas. O en el caso de que haya bastante volumen de tráfico, los paquetes que lleguen no deberían de tener gran cantidad de retardos de forma que de tiempo a que salgan los paquetes antes de que se empiecen a saturar las memorias, siendo de esta forma capaz de poder funcionar correctamente en el caso de que lleguen a tiempo antes de sobrescribir las memorias en paquetes que aún no han dado tiempo a salir.

Por lo tanto, como se pudo observar en la tabla 3.1, como máximo se podrán guardan en memorias BRAMs una cantidad de 9 ms (en el mejor de los casos con paquetes de 46 bytes de payload), una cantidad algo pequeña, por lo tanto las propuestas de trabajos futuros se estudiará poder usar memorias externar para poder guardar los paquetes.

En segundo lugar tenemos las limitaciones que conlleva el core HLS_array_menor, ya que no será lo mismo tener que ordenar unos pocos paquetes que tener que llegar a ordenar todo ya que el coste computacional cambiará bastante. Haciendo uso de la Figura 3.18 se podrá hacer una idea de la latencia que podrá tener este bloque según los casos mencionados. Y teniendo en cuenta que se trabaja a la frecuencia de 100 MHz, saber el tiempo de demora que acarreará.

Para empezar, se pone el caso en el que apenas hay que ordenar paquetes en el core, lo que habrá que hacer será multiplicar las latencias de la Figura 3.18, luego 1*3*1 nos dará una latencia de 3 ciclos de reloj, que serán unos 30nS (3/(100MHz)). En el peor de los casos, en el que se tengan que ordenar todos los paquetes que tengamos en nuestras memorias se tendría una latencia de 10500, siendo en tiempo 100,5uS (1050/(100MHz)). Este último caso es un caso absurdo y e improbable.

Entre casos intermedios, se puede analizar más o menos algún caso como el tiempo que tardaría en ordenar unos 10 paquetes, en ese caso la latencia sería de 23 (1*23*1) y tardando un tiempo de 0,23uS, para 20 paquetes, la latencia sería de 33 y siendo el tiempo de 0,33uS y por último caso, para unos 100 paquetes la latencia sería de 113 con un tiempo de 1,13us.

Respecto a la primera limitación que se han comentado, se requiere analizar la posibilidad de guardar los paquetes en memorias externas, teniendo dichas memorias mayor espacio que las BRAMs.

La segunda limitación requiere un rediseño de la estructura de reordenamiento que se plantea para futuros trabajos.

4.2 Uso de recursos FPGA

En esta sección se discute sobre el uso que hará cada IP-core de recursos en la FPGA [Xil05c]. Como se comentó en el estado del arte, es un componente estándar reprogramable por el usuario, en el que cada core se asignará una serie de componentes de la FPGA para llevar a cabo su función.

Mediante la herramienta Vivado_HLS [Xil15a], se obtendrá un reporte del número de componentes en la FPGA que se requerirá en cada core. A la hora de crear una solution en dicha herramienta, habrá que indicar que tipo de placa se usará. En este caso, como se comentó en la sección 3.7, será una Virtex 7 xc7vx690t.

En primer lugar tenemos el core HLS_input_pkt, en la figura a continuación se muestra el siguiente reporte.

Summary

Name	BRAM_18K	DSP48E	FF	LUT
Expression	-	-	0	291
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	1	-	0	0
Multiplexer	-	-	-	157
Register	-	-	413	-
Total	1	0	413	448
Available	2940	3600	866400	433200
Utilization (%)	~0	0	~0	~0

Tabla 4.1: Utilización estimada del core HLS_input_pkt

Como se puede observar, este módulo usará 413 FlipFlops para registros, 448 LUTs en multiplexores y expression y 1 BRAM para memorias.
El porcentaje de utilización en la FPGA, como se puede observar es muy bajo ya que no llega al 1% de utilización.

El siguiente módulo es HLS_array_menor, que muestra el siguiente reporte.

Summary

Name	BRAM_18K	DSP48E	FF	LUT
Expression	-	-	0	607
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	2	-	0	0
Multiplexer	-	-	-	290
Register	-	-	471	-
Total	2	0	471	897
Available	2940	3600	866400	433200
Utilization (%)	~0	0	~0	~0

Tabla 4.2: Utilización estimada del core HLS_array_menor

Este módulo requiere casi el doble de LUTs que el anterior, 897(290 para multiplexores y 607 para expression), por otro lado algún FlipFlop más, 471 todos ellos dedicados para registros y dos BRAMs para memorias. A pesar de ello, el porcentaje de utilización sigue siendo menor de 1% para este módulo en la FPGA.

Por último, para el módulo HLS_output_pkt, se tendrá el siguiente reporte.

Summary

Name	BRAM_18K	DSP48E	FF	LUT
Expression	-	-	0	439
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	-	-	-	-
Multiplexer	-	-	-	336
Register	-	-	330	-
Total	0	0	330	775
Available	2940	3600	866400	433200
Utilization (%)	0	0	~0	~0

Tabla 4.3: Utilización estimada del core HLS_output_pkt

En este caso, se harán uso de 330 FlipFlops para registros, 775 LUTs donde 439 de ellas serán para Expressions y 336 para multiplexores. En este core no se usarán BRAMs. El porcentaje de utilización de este core seguirá siendo menor del 1%.

La utilización de elementos en la FPGA [Xil05c] es bajo y lo que hará que crezca será el número de BRAMs que se utilicen para almacenamiento.

La implementación final utilizará estos tres cores más las memorias externas para almacenamiento de paquetes e índices según la figura 3.2. La cantidad de memoria BRAM utilizada depende del dispositivo siguiendo los resultados presentados en la sección 3.7.

5. Conclusiones y futuros trabajos

5.1 Conclusiones

A lo largo de este proyecto, se ha llevado a cabo un primer prototipo descripto en lenguaje de alto nivel para hardware, para llevar a cabo la funcionalidad equivalente a la aplicación Netem [ESPE13] en dispositivos reconfigurables. El objetivo perseguido con la implementación hardware de Netem es obtener altas precisiones en redes multiGbps, las que no son alcanzables con soluciones software.

La tecnología destino de esta implementación son circuitos reprogramables tipo FPGAs por cuestiones de coste y performance. Esta tecnología permite trabajar a altas velocidades pero con una moderada cantidad de memoria interna.

De cara a mejorar la productividad en el diseño se ha desarrollado el prototipo usando lenguaje de alto nivel para el uso de HLS (High Level Synthesis) mencionado en la sección 2.4 mediante la herramienta Vivado_HLS [Xil15a]. Esta alternativa redujo considerablemente el esfuerzo de diseño respecto al uso tradicional de lenguaje de descripción hardware (HDL) como Verilog o VHDL.

Tras una primera arquitectura funcional, se realizaron una serie de mejoras para poder optimizar la latencia interna de procesamiento y poder soportar tasas de línea a 10 Gbps. Los resultados preliminares muestran que se puede trabajar a tasa de línea con cargas moderadas del enlace.

Este primer prototipo sería capaz de funcionar en una FPGA simulando tráfico de red unos pocos micro segundos debido a las limitaciones de almacenamiento interno (BRAMs) presente en los dispositivos actuales. El uso de memorias externas se presenta como optimizaciones futuras.

El prototipo fue validado a nivel funcional utilizando simulaciones RTL (Register Transfer Level) utilizando trazas PCAP de pequeño y moderado tamaño. Adicionalmente la síntesis HLS muestra una muy baja ocupación de recursos lógicos en la FPGA siendo el límite real la cantidad de memoria interna.

5.2 Trabajo futuro

Las principales mejoras al trabajo actual surgen de las discusiones de la sección 4.1 donde discutieron sobre las limitaciones del primer prototipo.

La limitación sobre el escaso espacio en memorias BRAMs se puede resolver mediante memorias externar pero hay que tener en cuenta que las latencias de acceso son mucho más altas y debe ser contemplado.

Las latencias internas actuales no permiten trabajar a 10 Gbps en casos donde el desorden de los paquetes sea excesivo. Esta limitación sobre el módulo que debe ordenar los paquetes requiere un rediseño de la estructura de reordenamiento con el objetivo de mejorar estas latencias.

Otra de las tareas futuras es la implementación efectiva del prototipo en una tarjeta basada en FPGAs y su validación en el campo de aplicación.

Referencias:

[Xil15a] Xilinx Inc, Vivado Design Suite User Guide- High-Level Synthesis (UG902) June 2015. Available at: www.xilinx.com

[Xil15b] Xilinx Inc, Vivado Desing Suite User Guide - Getting Started September 30, 2015. Available at: www.xilinx.com

[Xil05c] Xilinx Inc, Using Block Rams in Spartan-3 Generation FPGAs, March 1, 2005. Available at: www.xilinx.com

[Wik15] 10 Gigabit Ethernet. Available at: www.wikipedia.com

[Lin15] The Linux Foundation, **NetEM** Network Emulation functionality for testing protocols, Available at: http://www.linuxfoundation.org/ collaborate /workgroups/ networking/ netem

[Kri15] Julio Kriger. Un nuevo protocolo de transporte SCTP-RR. Febrero 2015. Available at www.slideplayer.com

[Syn97] Synario, VHDL Reference Manual – March 1997. Available at www.ics.uci.edu

[Ash06] Peter Ashenden (Autor), The Designer's Guide to VHDL (Systems on Silicon) (Inglés) de Morgan Kaufmann; Edición: 3rd revised edition. 2006

[Ovi96] Verilog-A, Language Reference Manual - Analog Extensions to Verilog HDL August 1996. Open Verilog International. Available at www.accellera.org

[Qui15] C. Quintáns. Plataforma hardware para el autoaprendizaje de las FPGA y sus aplicaciones. Available at www.e-spacio.uned.es

[Kal12] Georgina Kalogeridou, NetFPGA-publicGetting Started Guide. May 12. Available at www.github.com

[Dig15] Digilent. Inc, NetFPGA-SUME Reference Manual. April 6, 2015. Available at www.digilentinc.com/sume

[Cai14] The CAIDA Anoymized Internet Traces 2014 Dataset. 2014 Available at www.caida.org